Pytest Fixtures

Pedro Paulo Monteiro Muniz Barbosa

Test Structure 3A:

1. **Arrange** - Create some objects;
2. **Act** - Simulate them
3. **Assert** - Check the results

Test-Driven Development By Example, Kent Beck - Pg. 99

Test Structure 3A:

```python


def example_1(num):
    return 1

```

```python

def test_example():
    parameter = 1                   # -> Arrange
    expected = 1                    # -> Arrange
    result = example_1(parameter)   # -> Act
    assert result == expected       # -> Assert

```

Test Structure 3A: One Step Test

```
12  def test_example():
13      assert example_1(1) == 1       # -> Arrange, Act, Assert
14
```

"If we want to remove duplication from our model code, do we want to remove it also from our test code? Maybe.."

Kent Beck, Test Driven-Development By Example - Pg. 163

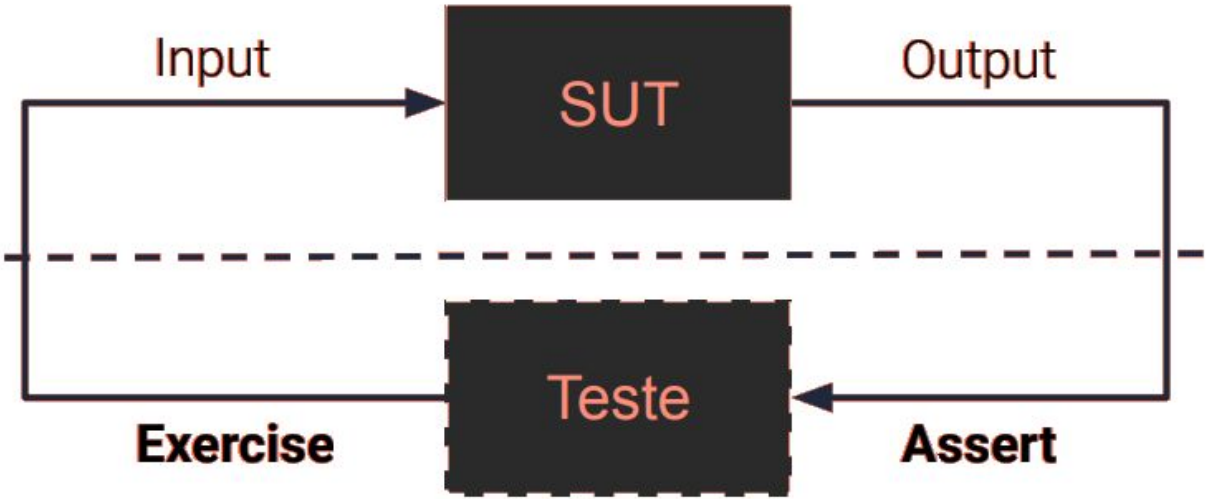
What is a fixture:

"Fixtures" are functions, which will run before each test function to which it is applied. Fixtures are used to feed some data to the tests such as database connections, URLs to test and some sort of input data. So, instead of running the same code for every test, we can attach fixture function to the tests and it will run and return the data to the test before executing each test.

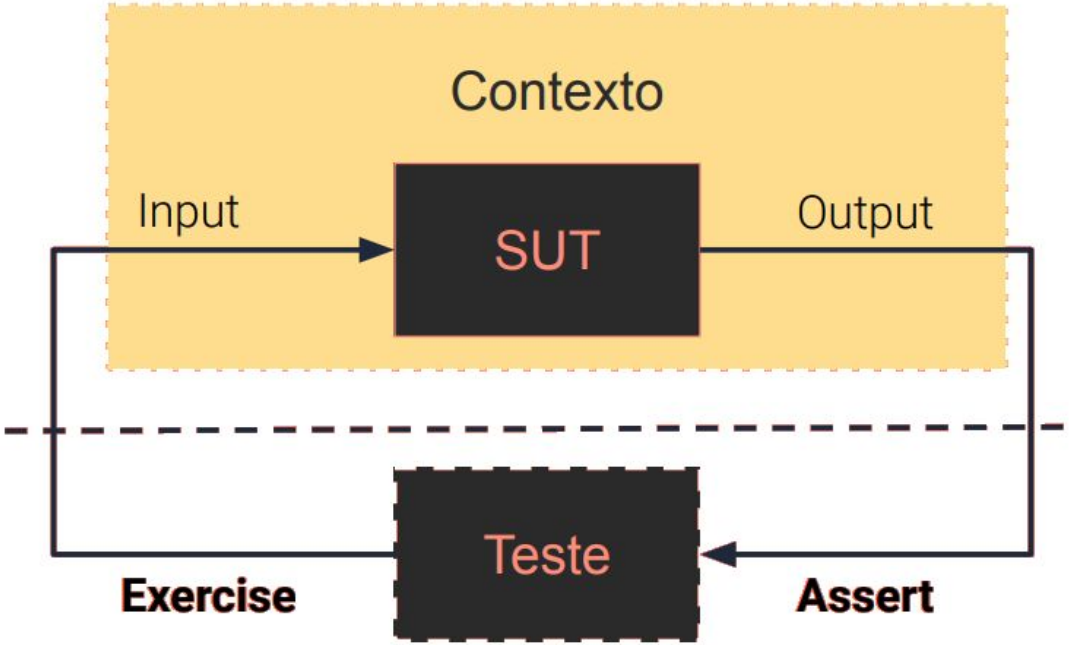
Four-Phases Test (Meszaros):

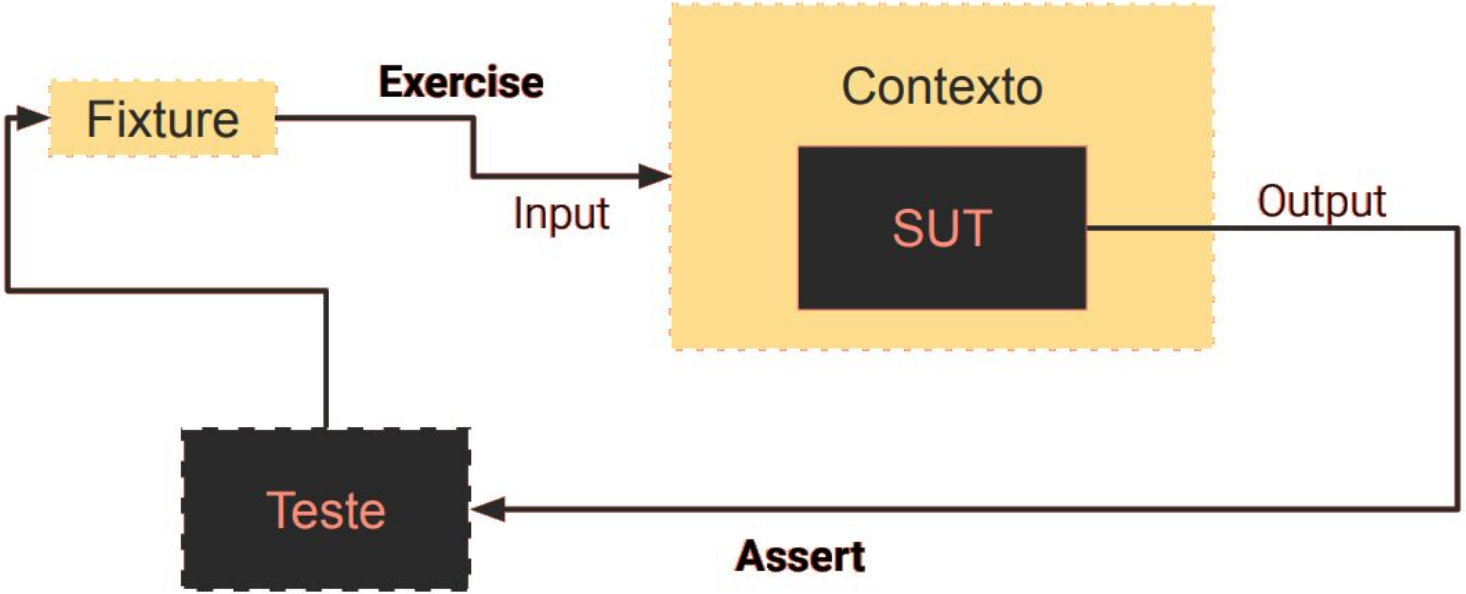
1. **Setup:** In the first phase, we set up the test fixture that is required for the SUT to exhibit the expected behavior as well as anything you need to put in place to be able to observe the actual outcome;
2. **Exercise:** In the second phase, we interact with the SUT;
3. **Verify:** In the third phase, we do whatever is necessary to determine whether the expected outcome has been obtained;
4. **TearDown:** In the fourth phase, we tear down the test fi xture to put the world back into the state in which we found it.

Testes usuais:



Testes utilizando fixtures:




Fixture
Exercise
Input
Contexto
SUT
Output
Teste
Assert

```python
import pytest


class Fruit:
    def __init__(self, name):
        self.name = name

    def __eq__(self, other):
        return self.name == other.name


@pytest.fixture
def my_fruit():
    return Fruit("apple")


@pytest.fixture
def fruit_basket(my_fruit):
    return [Fruit("banana"), my_fruit]


def test_my_fruit_in_basket(my_fruit, fruit_basket):
    assert my_fruit in fruit_basket
```

```
============================================================================ 1 passed in 0.17s =============================================================================
PS C:\Users\Pedro Paulo\fixtures_presentation\code> pytest tests_example_fixture.py -vv -s
============================================================================ test session starts =============================================================================
platform win32 -- Python 3.10.0, pytest-6.2.5, py-1.11.0, pluggy-1.0.0 -- C:\Users\Pedro Paulo\AppData\Local\Programs\Python\Python310\python.exe
cachedir: .pytest_cache
rootdir: C:\Users\Pedro Paulo\fixtures_presentation\code
plugins: Faker-12.0.0, factoryboy-2.1.0
collected 1 item

tests_example_fixture.py::test_my_fruit_in_basket PASSED

============================================================================ 1 passed in 0.14s =============================================================================
PS C:\Users\Pedro Paulo\fixtures_presentation\code>
```

Autouse:

Sometimes you may want to have a fixture (or even several) that you know all your tests will depend on. “Autouse” fixtures are a convenient way to make all tests automatically request them. This can cut out a lot of redundant requests, and can even provide more advanced fixture usage (more on that further down).

We can make a fixture an autouse fixture by passing in autouse=True to the fixture’s decorator.

```python
import pytest


@pytest.fixture
def first_entry():
    return "a"


@pytest.fixture
def order(first_entry):
    return []


@pytest.fixture(autouse=True)
def append_first(order, first_entry):
    return order.append(first_entry)


def test_string_only(order, first_entry):
    assert order == [first_entry]


def test_string_and_int(order, first_entry):
    order.append(2)
    assert order == [first_entry, 2]
```

```
============================================================================ 2 passed in 0.14s =============================================================================
PS C:\Users\Pedro Paulo\fixtures_presentation\code> pytest tests_example_fixture.py -vv -s
============================================================================ test session starts =============================================================================
platform win32 -- Python 3.10.0, pytest-6.2.5, py-1.11.0, pluggy-1.0.0 -- C:\Users\Pedro Paulo\AppData\Local\Programs\Python\Python310\python.exe
cachedir: .pytest_cache
rootdir: C:\Users\Pedro Paulo\fixtures_presentation\code
plugins: Faker-12.0.0, factoryboy-2.1.0
collected 2 items

tests_example_fixture.py::test_string_only ['a']
PASSED
tests_example_fixture.py::test_string_and_int ['a', 2]
PASSED

============================================================================ 2 passed in 0.14s =============================================================================
PS C:\Users\Pedro Paulo\fixtures_presentation\code>
```

TearDown:

When we run our tests, we'll want to make sure they clean up after themselves so they don't mess with any other tests (and also so that we don't leave behind a mountain of test data to bloat the system). Fixtures in pytest offer a very useful teardown system, which allows us to define the specific steps necessary for each fixture to clean up after itself.

TearDown - yield:

“Yield” fixtures yield instead of return. With these fixtures, we can run some code and pass an object back to the requesting fixture/test, just like with the other fixtures. The only differences are:

- The *return* is exchanged for the *yield*.
- Any TearDown code for this fixture is placed after the *yield*.

Once pytest figures out a linear order for the fixtures, it will run each one up until it returns or yields, and then move on to the next fixture in the list to do the same thing.

Once the test is finished, pytest will go back down the list of fixtures, but in the reverse order, taking each one that yielded, and running the code inside it that was after the yield statement.

As a simple example, let's say we want to test sending email from one user to another. We'll have to first make each user, then send the email from one user to the other, and finally assert that the other user received that message in their inbox. If we want to clean up after the test runs, we'll likely have to make sure the other user's mailbox is emptied before deleting that user, otherwise the system may complain.

```python
class MailAdminClient:
    def create_user(self):
        return MailUser()

    def delete_user(self, user):
        # do some cleanup
        pass

class MailUser:
    def __init__(self):
        self.inbox = []

    def send_email(self, email, other):
        other.inbox.append(email)

    def clear_mailbox(self):
        self.inbox.clear()


class Email:
    def __init__(self, subject, body):
        self.subject = subject
        self.body = body
```

```python
import pytest
from emaillib import Email, MailAdminClient

@pytest.fixture
def mail_admin():
    return MailAdminClient()


@pytest.fixture
def sending_user(mail_admin):
    user = mail_admin.create_user()
    yield user
    mail_admin.delete_user(user)


@pytest.fixture
def receiving_user(mail_admin):
    user = mail_admin.create_user()
    yield user
    mail_admin.delete_user(user)


def test_email_received(receiving_user, sending_user):
    email = Email(subject="Hey!", body="How's it going?")
    sending_user.send_email(email, receiving_user)
    assert email in receiving_user.inbox
```

```
============================================================================ 1 passed in 0.18s =============================================================================
PS C:\Users\Pedro Paulo\fixtures_presentation\code> pytest tests_example_fixture.py -vv -s
============================================================================ test session starts =============================================================================
platform win32 -- Python 3.10.0, pytest-6.2.5, py-1.11.0, pluggy-1.0.0 -- C:\Users\Pedro Paulo\AppData\Local\Programs\Python\Python310\python.exe
cachedir: .pytest_cache
rootdir: C:\Users\Pedro Paulo\fixtures_presentation\code
plugins: Faker-12.0.0, factoryboy-2.1.0
collected 1 item

tests_example_fixture.py::test_email_received
Creating the receiving user
<emaillib.MailUser object at 0x000002DB2F8A2D10>
Creating the sending user
<emaillib.MailUser object at 0x000002DB2F928340>
PASSED
Deleting the sending user
Deleting the receiving user
```

Fixtures Scopes:

Fixtures are created when first requested by a test, and are destroyed based on their scope:

- `function`: the default scope, the fixture is destroyed at the end of the test.
- `class`: the fixture is destroyed during teardown of the last test in the class.
- `module`: the fixture is destroyed during teardown of the last test in the module.
- `package`: the fixture is destroyed during teardown of the last test in the package.
- `session`: the fixture is destroyed at the end of the test session.

Disponibilidade da Fixture:

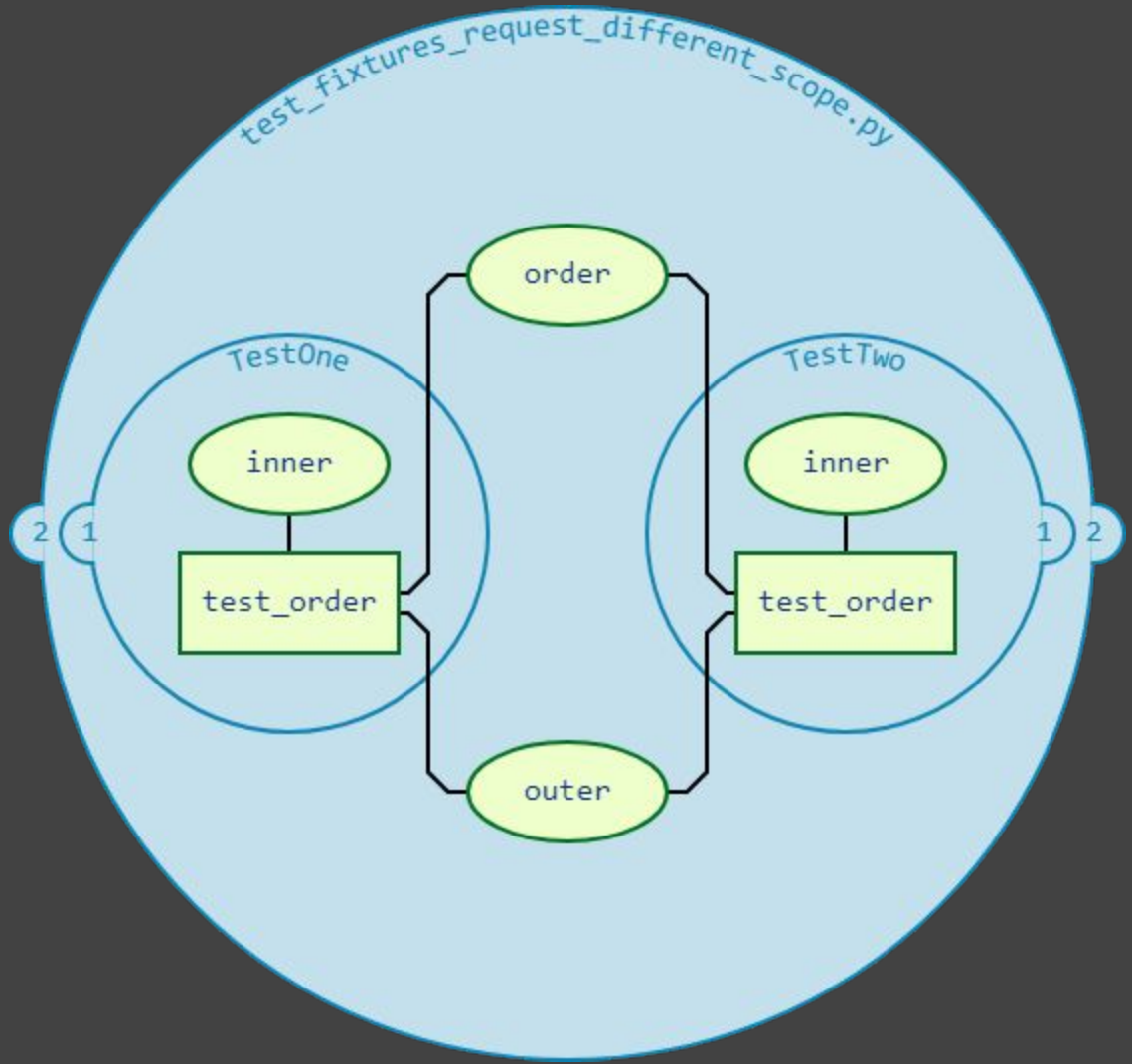
Fixture availability is determined from the perspective of the test. A fixture is only available for tests to request if they are in the scope that fixture is defined in. If a fixture is defined inside a class, it can only be requested by tests inside that class. But if a fixture is defined inside the global scope of the module, than every test in that module, even if it's defined inside a class, can request it.

In the same way, a test can also only be affected by an autouse fixture if that test is in the same scope that autouse fixture is defined in.

A fixture can also request any other fixture, no matter where it's defined, as long as the test requesting them can see all fixtures involved.

```python
import pytest

@pytest.fixture
def order():
    return []

@pytest.fixture
def outer(order, inner):
    order.append("outer")

class TestOne:
    @pytest.fixture
    def inner(self, order):
        order.append("one")

    def test_order(self, order, outer):
        assert order == ["one", "outer"]

class TestTwo:
    @pytest.fixture
    def inner(self, order):
        order.append("two")

    def test_order(self, order, outer):
        assert order == ["two", "outer"]
```

```
============================================================================ 2 passed in 0.15s =============================================================================
PS C:\Users\Pedro Paulo\fixtures_presentation\code> pytest tests_example_fixture.py -vv -s
============================================================================ test session starts =============================================================================
platform win32 -- Python 3.10.0, pytest-6.2.5, py-1.11.0, pluggy-1.0.0 -- C:\Users\Pedro Paulo\AppData\Local\Programs\Python\Python310\python.exe
cachedir: .pytest_cache
rootdir: C:\Users\Pedro Paulo\fixtures_presentation\code
plugins: Faker-12.0.0, factoryboy-2.1.0
collected 2 items

tests_example_fixture.py::TestOne::test_order PASSED
tests_example_fixture.py::TestTwo::test_order PASSED

============================================================================ 2 passed in 0.15s =============================================================================
PS C:\Users\Pedro Paulo\fixtures_presentation\code>
```


test_fixtures_request_different_scope.py
order
TestOne
inner
2
1
test_order
TestTwo
inner
1
2
test_order
outer

Fixtures are defined using the @pytest.fixture decorator. Pytest has useful built-in fixtures, listed down below for reference:

capfd
capfdbinary
caplog
capsys
capsysbinary
cache
doctest_namespace
monkeypatch
pytestconfig
record_property
record_testsuite_property
recwarn
request
testdir
tmp_path
tmp_path_factory
tmpdir
tmpdir_factory

References:

MESZAROS, Gerard. *xUnit Test Patterns*. Addison-Wesley, 2007.

BECK, Kent. *Test Driven-Development By Example*. Addison-Wesley, 2000.

pytest fixtures: explicit, modular, scalable. pytest, Full pytest Documentation, 2015. Available in: <https://docs.pytest.org/en/6.2.x/fixture.html#what-fixtures-are>. Accessed on: February 2, 2022